A Black-Box Tracing Technique to Identify Causes of Least-Privilege Incompatibilities

Shuo Chen
University of Illinois at Urbana-Champaign
shuochen@uiuc.edu

John Dunagan   Chad Verbowski   Yi-Min Wang
Microsoft Research
{jdunagan, chadv, ymwang}@microsoft.com

Abstract

Most Windows users run all the time with Admin privileges. This significantly increases the vulnerability of Windows systems because the compromise of any user-level application becomes a system compromise. To address this problem, we present a novel tracing technique to identify the causes of least-privilege incompatibilities *(i.e., application dependencies on Admin privileges). Our evaluation on a number of real-world applications shows that our tracing technique significantly helps developers fix least-privilege incompatibilities, and can also help system administrators mitigate the impact of least-privilege incompatibilities through local system policy changes.*

1 Introduction

The principle of least-privilege is that software should run only with the privileges necessary to accomplish the task at hand. Much previous work has sought to build software better conforming to this principle (e.g., [6, 18, 26]). Unfortunately, adherence to the least-privilege principle on Windows systems is quite low: Most users run all the time as members of the Administrators group ("with Admin privileges"), similar to executing all commands as root on a UNIX system. This increases the severity of security threats faced by Windows users, because the compromise of any user application becomes a system compromise. This threat is both acute and widespread; attacks against user level networking applications are common, and include spyware [29, 33], self-propagating email [7], web browser exploits [9, 10], and instant messaging (IM) client exploits [8].

Many Windows users run with Admin privileges because, in fact, the applications they use require Admin privileges. A Microsoft online article lists 188 such applications with least-privilege incompatibilities [21]. Furthermore, this problem spans many user types: children that play *Bob the Builder*; anyone filing taxes with *TurboTax*; corporate employees that connect to their corporate network using *Remote Access Service*; and developers that use *Razzle* to setup their build environment.

In addition to causing individual applications to require Admin privileges, least-privilege incompatibilities exert a social pressure to run all applications with Admin privileges. This social pressure exists for two reasons: First, least-privilege incompatible applications often fail with misleading error messages, so users without Admin privileges spend significantly more time troubleshooting; Second, the number of least-privilege incompatible applications is sufficiently great that starting each one from a separate account with Admin privileges, or setting up scripts to do this semi-automatically, is a significant inconvenience.

In this paper, we describe a tracing technique to identify the causes of least-privilege incompatibilities, making it easier to fix or mitigate them. Our technique is black-box, i.e. it does not require source code. Identifying the causes of least-privilege incompatibilities enables two important scenarios:

- **Developers can fix least-privilege incompatibilities more easily.** Developers face a number of challenges in identifying and understanding least-privilege incompatibilities. In large software projects, developers must often modify code written by others, where they have no a priori insight into failing security checks. Additionally, libraries and other software components commonly encapsulate system calls and are sometimes available only in binary form, making failures more opaque. Simply setting breakpoints often requires many iterations to narrow down the source code line responsible for a single least-privilege incompatibility. In contrast, our tracing technique allows a single test pass to produce a list of all the least-privilege incompatibilities in the exercised code paths, and to provide additional information beyond source lines, such as object names, Access Control Lists (ACLs), and call stacks (if symbol files are present). Our evaluation suggests that this information significantly reduces the total time required to fix least-privilege incompatibilities.

- **System administrators can mitigate some least-privilege incompatibilities through system policy changes.** Making ACL changes so that applications can run with reduced privilege is a well-known technique [6, 23]. Our tracing technique enables faster identification of both the relevant ACLs and other causes of least-privilege incompatibilities, such as missing named privileges. Our evaluation suggests that this often allows system administrators to modify the system policy so that previously least-privilege incompatible applications can be run without Admin privileges.

The use of tracing, a dynamic technique, implies a standard set of tradeoffs. Our tracing technique only identifies least-privilege incompatibilities on exercised code paths, and achieving good code coverage may require the additional use of sophisticated test generation technology [3, 16, 19]. Because tracing can miss least-privilege incompatibilities on unexercised code paths, it does not have perfect completeness. In Section 5, we discuss the reasons to prefer tracing to static techniques, even though static techniques can provide perfect completeness, for the particular problem of identifying least-privilege incompatibilities.

Our evaluation on eight real-world applications demonstrates the accuracy and usefulness of our tracing technique. Because our technique does not require source code, we were able to include in our evaluation third-party applications for which we only have binaries. To demonstrate accuracy, we show that few logged security checks were unrelated to least-privilege incompatibilities (good soundness), and that bypassing the remaining logged checks allows the application to run without Admin privileges (an empirical test of completeness). To demonstrate usefulness, we first show that the number of security checks responsible for least-privilege incompatibilities is small. Based both on the traced least-privilege incompatibilities and consultations with developers knowledgeable about the applications, we conclude that this information is a significant help in fixing the incompatibilities.

The remainder of this paper is organized as follows: Section 2 provides relevant background information on the Windows Security Model. Section 3 describes our implementation. Section 4 presents our evaluation. Section 5 discusses related work. Section 6 concludes and discusses our plans for future work.

2 Background on the Windows Security Model

We describe the abstractions and mechanisms of the Windows security model by comparison to the UNIX security model. A Windows *token* represents the security context of a user. Tokens are inherited by processes created by the user. A token contains multiple Security IDs (*SIDs*), one expressing the user's identity, and the rest for groups that the user belongs to, such as the Administrators group, or the Backup Operators group. UNIX similarly attaches both a user ID and a set of group IDs to a process. In order to implement the setuid mechanism, UNIX adds another two user IDs, so that at any point there is a real user ID, an effective user ID, and a saved user ID [11].

Windows does not support the notion of a setuid bit, and Windows developers typically follow a different convention in implementing privileged functionality. For example, in UNIX, *sendmail* was historically installed with the setuid bit so that an unprivileged user could invoke it, and the process could then read and write to the mail spool, a protected OS file. In Windows, a developer would typically write sendmail as a service (equivalent to a UNIX daemon), and a user would interact with sendmail using Local Procedure Call (LPC). One would implement the sendmail command-line interface as a simple executable that sends the command line arguments to the service via LPC. The Windows service model allows services to be started on demand, so dormant services occupy no memory, just as in the UNIX sendmail case.

A Windows token also contains a set of *privileges* (which can be enabled or disabled), such as the SystemTime or Shutdown privilege. These two privileges grant the abilities, respectively, to change the system clock and to shutdown the system. Conceptually, privileges are used to grant abilities that do not apply to a particular object, while accesses to individual objects are regulated using Access Control Lists (*ACLs*). In contrast, UNIX typically uses groups to implement named privileges. For example, membership in the floppy group grants access to the floppy drive. To create an equivalent to the SystemTime privilege in UNIX, one might create a SystemTime group, create a ChangeSystemTime setuid executable, set its group to SystemTime, and give it group-execute permission.

Windows and UNIX both support ACLs, but again, their implementations are slightly different. UNIX file systems typically associate each file with an owner and a group, and store access rights for the owner, members of the group, and all others. Windows ACLs can contain many <SID, access> pairs, as in AFS (the Andrew File System). These <SID, access> pairs are used to grant one user the ability to read and write the object, another user the ability only to read the object, all members of another group the ability to read the object, etc. ACLs in Windows can be attached not only to files, but to any object accessible through a handle, such as registry entries and semaphores. In UNIX, and more so in Plan 9, access control is made uniform across resources by exporting most resources through the file system (e.g., /dev/audio).

2.1 Security Checking Functions

The interface to security checking in Windows is complex. We identified a small set of security checking functions to instrument, and we took several steps to assure ourselves of their completeness: reading the Windows source code, consulting a senior Windows architect, and examining the kernel call stack at observed application failures. Based on these steps and our success identifying least-privilege incompatibilities in the applications in our evaluation section, we have good confidence in the set of functions we identified. Note that the complexity of security interfaces is not unique to Windows; previous work has discussed the subtleties of UNIX security interfaces[11, 14].

The five functions we identified, and their role in the security subsystem, are presented in Figure 1: the functions themselves are circled, and the arrows denote function inputs and outputs. For the purpose of discussion, we have changed the function names to make them more intelligible. *Privilege-Check* is used to check that privileges are held and enabled in the token. *Adjust-Privilege* is used to enable or disable privileges. *Access-Check* is used to check whether a user has access to a particular object, as determined by its ACL. *Reference-Object* also performs access checks; requests to read or write an object flow through this function, which checks the Handle Table to see whether the ability to perform the operation was previously granted by *Access-Check* when the handle to the object was created.

SID-Compare is used both internally by the security subsystem and directly by applications. In particular, least-privilege incompatible applications often use *SID-Compare* to fail early. The application checks if the user holds a SID granting membership in the Administrators group, and fails if not. Intercepting this direct application check was necessary for us to determine the later (and more interesting) set of checks causing least-privilege incompatibilities. Of course, a developer attempting to fix a least-privilege incompatible application would find removing this *SID-Compare* check to be an obvious modification.

3 Identifying Least-Privilege Incompatibilities

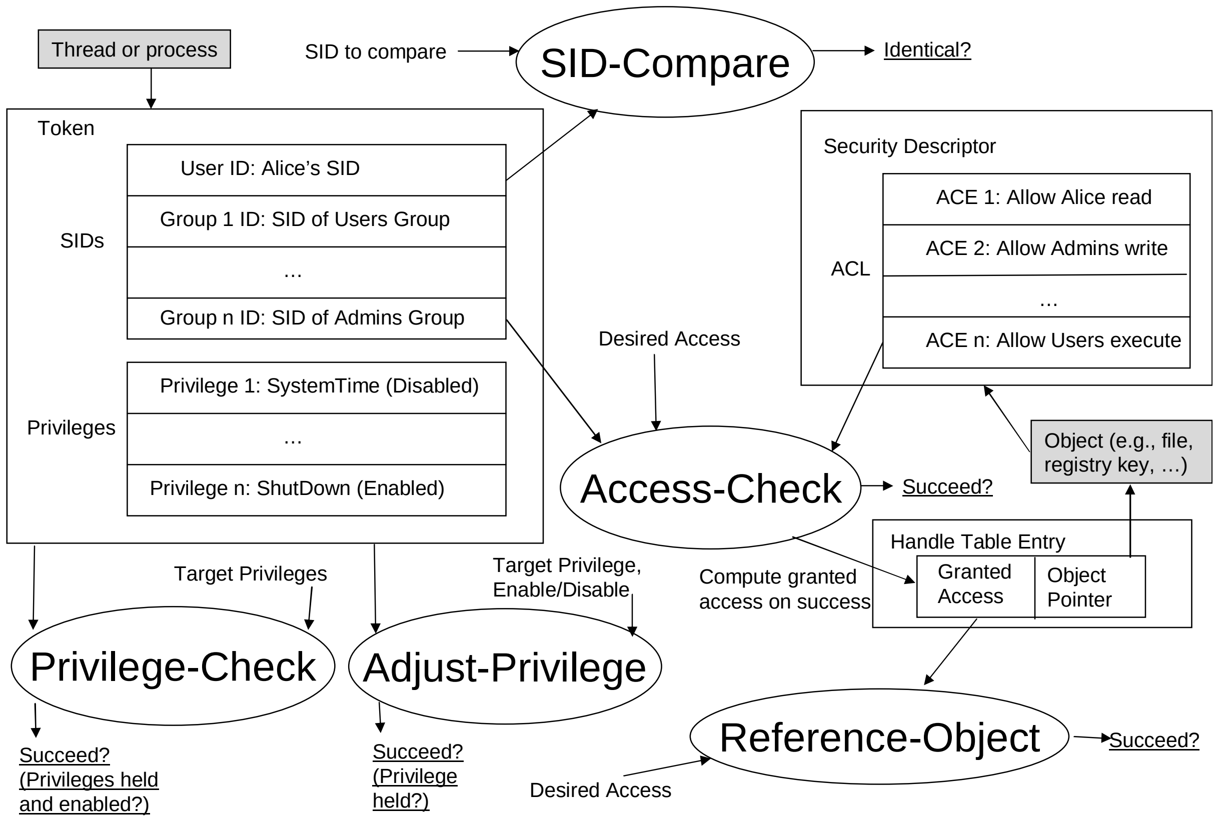
We implemented our tracing technique for identifying least-privilege incompatibilities by adding two components to the Windows XP Service Pack 1 kernel, a Security Check Monitor and Noise Filter and a Security Check Event Logger. Because security checks are a tiny fraction of an unmodified system's performance, the overhead our components added to each security check had a negligible effect on overall system performance. To apply our technique, a developer or system administrator starts the tracer, runs the incompatible application with Admin privileges, and then stops the tracer. While the tracer is running, the Security Check Monitor and Noise Filter component applies a conservative noise filtering algorithm to keep only those security checks that might be responsible for least-privilege incompatibilities. The actual logging of these checks is done the Security Check Event Logger component. After tracing, we apply a separate log validation step, described in more detail in Section 3.2. Figure 2 shows this workflow.

3.1 Security Check Monitoring and Noise Filtering

We developed a simple noise filtering algorithm specific to identifying least-privilege incompatibilities motivated by the following observation: large numbers of security checks fail on running Windows systems without any noticeable end-user impact. We speculate that these failed checks come from applications and libraries attempting to acquire object with rights they do not require for their proper functioning, but we have not yet managed to investigated this. Our noise filtering algorithm identifies calls that succeed with Admin privileges and fail without them. This has no false negatives, i.e., it does not eliminate any true least-privilege incompatibilities from the log. It does not entirely eliminate false positives: some applications attempt to acquire objects with rights they do not need, falling back to acquiring the object with fewer rights without any apparent adverse effect.

Our noise filtering algorithm assumes the user is running the application with Admin privileges. In the security subsystem, we intercept all security checks, and initially allow the check to pass through unmodified. If the check is successful and the token contained membership in the Administrators group, the noise filter temporarily removes this membership from the token and performs a second check. If this second check fails, the Security Check Event Logger is called. Although our implementation only differentiates between membership and non-membership in the Administrators group, it would be straightforward to configure the component to handle other groups (e.g., the Backup Operators group). To convince ourselves that this approach had merit, we performed a quick experiment, collecting three 2-hour traces during regular office hours on one of our primary machines. The results of these traces are summarized in Table 1.

In each of these traces, the set of security checks that would be logged after applying our noise filtering algorithm (the column labeled Difference) is much smaller than the total number of failed checks. The 2K-3K remaining failed checks still constitute a conservative superset of the checks corresponding to least-privilege incompatibilities. Though 2K-3K checks is probably too many to examine by hand, in practice we expect the tracer to be run in much shorter intervals — identifying the least-privilege incompatibilities described in Section 4 required trace lengths of less than



Thread or process
SID to compare
SID-Compare
Identical?
Token
User ID: Alice's SID
Group 1 ID: SID of Users Group
SIDs
…
Group n ID: SID of Admins Group
Privilege 1: SystemTime (Disabled)
Privileges
…
Privilege n: ShutDown (Enabled)
Security Descriptor
ACE 1: Allow Alice read
ACE 2: Allow Admins write
ACL
…
ACE n: Allow Users execute
Desired Access
Object (e.g., file, registry key, …)
Access-Check
Succeed?
Handle Table Entry
Granted Access
Object Pointer
Compute granted access on success
Target Privileges
Target Privilege, Enable/Disable
Privilege-Check
Adjust-Privilege
Reference-Object
Succeed?
Succeed? (Privileges held and enabled?)
Succeed? (Privilege held?)
Desired Access

Figure 1. *Windows Security Checking Functions*

	Security checks	Security checks with user token	Failures with Admin privileges	Failures without Admin privileges	Difference
Trace 1	1,756,000	417,257	79,317	81,597	2,280
Trace 2	1,124,000	315,014	64,336	66,385	2,049
Trace 3	913,000	422,783	94,453	97,170	2,717

Table 1. *Two-Hour Traces of Security Checks*

20 seconds. Manually inspecting the logs also yielded two other unsurprising observations. First, security checks tend to occur in bursts right after new processes are started. Second, the potential causes of least-privilege incompatibilities appear to cover the entire range of security checks: access check failures on semaphores and registry keys, privilege check failures, and many others.

This noise filtering algorithm depends on the fact that the underlying Windows security subsystem is stateless because it re-executes certain calls with modified arguments. A natural alternative approach to modifying the security subsystem would have been to modify the APIs that access resources, e.g. the file system or socket APIs. However, the stateful nature of these interfaces would have made our approach to noise filtering either difficult or impossible. For example, re-executing a File-Open call would have required closing the file, attempting to reopen it with a different set of permissions, and doing the appropriate fixup. Appropriately handling calls to arbitrary objects would have been even more challenging.

Intricacies of *Access-Check* with MAXIMUM_ALLOWED. The *Access-Check* function can be called either with an explicit list of desired accesses, such as read and write, or it can be called with a special argument (MAXIMUM_ALLOWED) asking for all allowed accesses. Because Admin privileges grant additional accesses on most objects, it might seem necessary to log most calls with MAXIMUM_ALLOWED as potential least-privilege incompatibilities. However, this would quickly lead to a large number of false positives: for example, when we started the TurboTax application (one of the examples in our evaluation), *Access-Check* was called

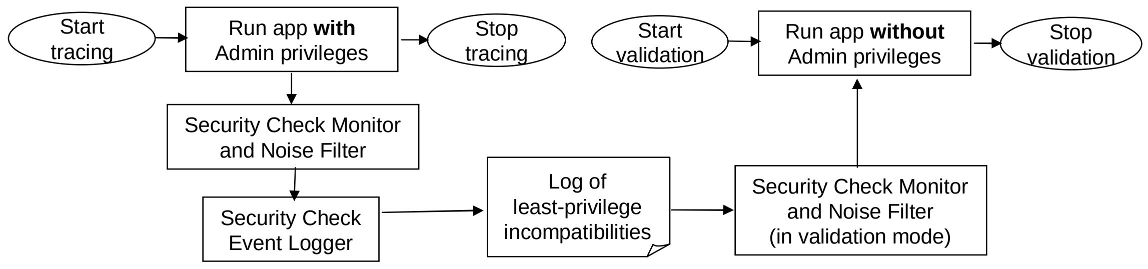

Start tracing
Run app **with** Admin privileges
Stop tracing
Security Check Monitor and Noise Filter
Security Check Event Logger
Log of least-privilege incompatibilities
Start validation
Run app **without** Admin privileges
Stop validation
Security Check Monitor and Noise Filter (in validation mode)

Figure 2. *Workflow of Tracing and Validation*

303 times with MAXIMUM_ALLOWED access, and 189 of these calls return different accesses based on whether a user has Admin privileges.

Fortunately, all the calls that we observed to *Access-Check* with MAXIMUM_ALLOWED occurred during object handle creation. When object handles are created, the accesses granted by *Access-Check* are cached, and later calls through *Reference-Object* are compared to these cached accesses. This allowed us to avoid the potentially large number of false positives due to MAXIMUM_ALLOWED by modifying the *Reference-Object* interface to incorporate our noise filtering algorithm: retry successful requests to see whether they would have succeeded without Admin privileges.

Applying the noise filtering algorithm at the *Reference-Object* interface required maintaining a small amount of additional state for each object handle. When object handles are created with *Access-Check* and MAXIMUM_ALLOWED, we initialize an additional field in the handle, AssumedGrantedAccess, with the results of *Access-Check* and MAXIMUM_ALLOWED after removing Admin privileges. When *Reference-Object* is later called with an object handle, the desired access is compared to both the actual granted accesses and AssumedGrantedAccess. The request is then logged if the desired access is allowed by the actual granted accesses and not by AssumedGrantedAccess.

3.2 Security Check Event Logger

We implemented the Security Check Event Logger by modifying ETW (Event Tracing for Windows), a kernel component that already allows logging events such as registry accesses, page faults and disk I/O. Each security check log entry indicates the current process name, the monitored security checking function, target privileges, the desired access and granted access, a stack dump (the return addresses on the kernel stack), and the object name.

Obtaining the object name of each *Access-Check* call is more difficult than obtaining the other information. *Access-Check* is performed on a security descriptor and a token. There is no backward pointer from the security descriptor to the object, and indeed, a security descriptor can be created by a programmer without reference to an object, though this practice is rare. To obtain object names when they exist, the logger walks back along the kernel stack, traversing frame pointer frames. The traversal stops at any function frame that is known to contain object name information, which is then written to the log. This technique requires a kernel compiled with frame pointers, which is the case for Windows XP Service Pack 1. We have currently implemented retrieving object names from five functions that we know to be particularly common parents of *Access-Check*, such as Create-File. This has been sufficient to give us very good coverage. It allowed us to debug all the least-privilege incompatibilities given in the evaluation section, and it returned object names for 98.3% of the access checks in one of our 2-hour traces (8490 out of 8639 checks).

3.3 Log Validation

We performed a separate validation step to confirm that all the least-privilege incompatibilities in the code paths we managed to exercise were detected by our tracing technique. We could not directly modify the code for some of the applications in our evaluation, so instead we modified the behavior of our Security Check Monitor and Noise Filter to change the outcomes of the security checks themselves. This validation step is the reverse of the tracing step: applications are run without Admin privileges, and previously logged checks are made to succeed where they otherwise would have failed. The application will work without Admin privileges if and only if the logged checks cover all causes of least-privilege incompatibilities.

Though it might seem possible to mitigate all least-privilege incompatibilities by using the validation technique to change the outcome of certain security checks (often effectively changing ACLs), we have several reasons to urge caution in this approach. First, some applications legitimately require Admin privileges, and the objects they are accessing should retain their strict ACLs. A significant complicating factor is that the ACLs may be for kernel objects whose significance is less clear than files or registry keys. Secondly, for applications that should not require Admin privileges, developers are free to change application behavior in many ways. Changing ACLs may require significantly more work than the alternative modifications to application behavior. Finally, the architecture of the Security Check Monitor and Noise Filter component requires a lookup table to determine whether to change the outcome of particular security checks. Reconfiguring ACLs in place has better scalability and efficiency, avoiding both the lookup step and the need to maintain additional state in the security subsytem. Indeed, this is exactly how ACL modifications are currently handled in the file system and registry.

4 Evaluation

We evaluated the effectiveness of our tracing technique on eight least-privilege failure scenarios drawn from real applications. These applications include small utility programs, video games, document processing applications, and software development tools, and span the spectrum of users, including pre-school children, teenagers, professionals and home users. The purpose of the evaluation is both to understand the effectiveness of the technique in producing a small set of security checks responsible for the least-privilege incompatibility, and to understand how helpful this would be to a developer seeking to fix the incompatibility, or a system administrator seeking to mitigate it.

For our experiments we installed and traced applications using an account with Admin privileges. We found that most least-privilege incompatibilities are encountered quickly: for interactive applications, we found that we only needed to trace application startup, while for non-interactive applications (e.g., scripts), we traced the entire run of the application. We validated our logs using a second account without Admin privileges. We found that the causes of least-privilege incompatibilities in our evaluation fall into three categories: overly-restrictive ACL settings, insufficient granularity of privilege in the application design, and programmatic enforcement of unnecessary privilege requirements.

4.1 Overly Restrictive ACLs

The three applications in this section required elevated privileges because they either stored their settings in a more secure location than necessary, or they did not correctly configure ACLs to allow access by the appropriate users. These problems are all fixable with small code changes, and it appears to be possible to work around them by manually reconfiguring the relevant ACLs.

4.1.1 Bob The Builder Game

“Bob The Builder, Can We Fix It” is a video game designed for children as young as 3. If a user attempts to start the game without Admin privileges, an error message appears stating “Automenu: insufficient privilege”. The tracer intercepted 4002 checks during application startup, of which 899 would have failed if the user had not had Admin privileges. Only 15 checks survived noise filtering. The 5 unique entries among these 15 checks are shown in Table 2.

Two things point to the first entry in the log being the likely cause of the least-privilege incompatibility. First, the error message mentions the AutoMenu process. Second, HKEY_LOCAL_MACHINE (HKLM) is a portion of the Registry used for storing machine-wide settings. We used our validation technique to confirm this hypothesis. Although we have not heard from the application developers directly, anecdotal evidence points to this being a common mistake leading to least-privilege incompatibility, easily fixed by using a per-user store. This least-privilege incompatibility is also simple to mitigate by modifying ACLs because no system critical information is stored in the “Bob the Builder” section of the Registry. We have not yet deduced why explorer also generates entries in the log, but as mentioned previously, our technique sometimes generates false positives.

4.1.2 RAZZLE

Several Microsoft products use the razzle build environment configuration tool. Developers must have Admin privileges to use the current version of this tool. When a user without Admin privileges runs razzle and then attempts to change to a source code directory, they receive a “permission denied” message.

Tracing razzle from start to finish yielded 7 log entries (shown in Table 3) out of 8660 security checks. One notices immediately from the trace that razzle launches a series of other processes. Our first hypothesis was that the ACL on c:\sysman (our source code directory) was responsible for the least-privilege incompatibility, and we changed the ACL manually. However, when we ran razzle a second time, the ACLs reverted to requiring Admin privileges. Our second hypothesis was that the child process razacl.exe was changing the ACLs. We confirmed this by changing the ACL manually and then removing the razacl executable; this allowed a user without Admin privileges to use razzle. We learned from consulting razzle developers that razacl removes user accounts from ACLs in the build tree to produce a consistent build environment across user accounts. Changing razzle to produce a consistent build environment without requiring Admin privileges is trivial, and indeed, the next version of razzle is already slated to have this change incorporated. This least-privilege incompatibility is easily mitigated by a system administrator because razzle is a shell script, and so a system administrator can easily modify it to not use razacl. If razzle had been compiled code that

Security Function	**Process**	**Object Name**
Reference-Object	Automenu.exe	\REGISTRY\HKLM\SOFTWARE\BBC Multimedia\Bob the Builder\1.0.0
Access-Check	explorer.exe	\Program Files\THQ\Bob the Builder\StartBTB.exe
Access-Check	explorer.exe	\WINDOWS\explorer.exe
Access-Check	explorer.exe	\WINDOWS\system32\mydocs.dll
Access-Check	explorer.exe	\WINDOWS\system32\shell32.dll

Table 2. *Unique Log Entries for Bob The Builder*

required razacl in order to complete, a system administrator still could have written a separate script to reset the ACLs after running razacl.

4.1.3 Microsoft Greetings 2001

Microsoft Greetings 2001 is a document processing application. Our trace of Microsoft Greetings' startup recorded 37 potential causes of least-privilege incompatibilities, (summarized in Table 4), out of the 12,618 total security checks.

In validating the logs, we found that the first three classes of logged security checks must succeed for the application to be usable without Admin privileges. All the security checks in the remaining two classes appear to be false positives. All the failed checks appear to be easily fixable, as they seem to reflect the standard mistake of storing settings in a machine-wide scope. Indeed, the next (and renamed) version of this software, Microsoft Picture It! 2002, does not have any least-privilege incompatibilities. From the system administrator perspective, the second and third classes of security checks are mitigatable using only standard tools, but the first class requires the use of our validation technique. Finally, the large number of least-privilege incompatibilities in this example illustrates the benefits of requiring only one trace to identify all the incompatibilities.

4.2 Insufficient Privilege Granularity in Application Design

The three applications in this section all have some functionality that is appropriate for all users, and some functionality that should only be usable with Admin privileges. However, they all fail to accommodate both modes of operation in their design, and consequently are not usable at all without Admin privileges. We thank [4] for bringing many of these examples to our attention.

4.2.1 Remote Access Service (RAS)

RAS is a program for corporate employees to remotely connect to the corporate network. Running RAS without Admin privileges leads to an error message roughly one minute after the program starts. Reproducing this problem was technically challenging because our lab did not allow us to easily fake the remote environment that RAS assumes. Our workaround was to trace a small script [22] that replicated the core RAS behavior, and then to validate the results using the real RAS program from a remote location.

Tracing the small script generated 7 log entries out of 2566 security checks. Six of the seven checks were related to files, registry keys and TCP/IP devices that we eliminated as causes of least-privilege incompatibilities using our validation technique. Causing just the last check to succeed allowed both the script and the real RAS program to be run without Admin privileges.

Analyzing this one security check in more detail, we saw the Windows script interpreter attempting to enumerate all network connections by calling the function *get_EnumEveryConnection* in class *CNetSharingManager* defined in *HNETCFG.dll* (Home Networking Configuration Manager). This function checks that the user has Admin privileges using the function *CheckTokenMembership* exported by *ADVAPI32.dll*, which internally calls into the kernel function *Access-Check*.

From discussions with the developers of this tool we learned that RAS enumerates all network connections, and switches all of them to run over the newly created Virtual Private Network (VPN). The only API for enumerating network connections enumerates them for all connections (not just the current user), and this API was designed to be usable only with Admin privileges. The RAS developers seem to be faced with two possible solutions to this least-privilege incompatibility: a Windows service could perform work on behalf of RAS, thus allowing any user to switch other users' connections to run over the VPN; alternatively, if an alternative API were present that allowed enumerating only the network connections for a particular user, one might be satisfied with switching only that user's connections to run over the VPN. A system administrator would be restricted to addressing this problem by using our validation technology to cause this one check for Admin privileges to succeed — this would be roughly equivalent to the first of the possible developer solutions.

This example illustrates that least-privilege incompatibilities may result from non-intuitive security checks where a library indirectly checks access in its lower level imple-

Security Function	Process	Object Name or Security Action
Access-Check	explorer.exe	\WINDOWS\system32\cmd.exe
Adjust-Privilege	razacl.exe	Enable Security privilege
Privilege-Check	razacl.exe	Check if Security privilege enabled
Access-Check	cmd.exe	\sysman
Access-Check	findstr.exe	\sysman
Access-Check	perl.exe	\sysman
SID-Compare	tfindcer.exe	Determine if user has Admin privileges

Table 3. *Log Entries for RAZZLE*

Security Function	Object Name or Security Action
SID-Compare	Determine if user has Admin privileges
Access-Check	\Program Files\Microsoft Picture It! PhotoPub\pidocob.dll
3 *Access-Check*	\REGISTRY\HKLM\SOFTWARE\Microsoft\Picture It! (and subkeys)
22 *Access-Check*	\REGISTRY\HKLM\SOFTWARE\Classes (and subkeys)
11 additional entries	...

Table 4. *Summary of Log Entries for Microsoft Greetings 2001*

mentation. In cases like these, even developers benefit from tracing not requiring source code. However, because we actually did have access to the application source code, we were able to construct the entire sequence of calls responsible for the least-privilege incompatibility, not just within the RAS application.

4.2.2 Windows Power Configuration

Windows power options are configured per user and stored in the user's profile. However, Admin privileges are required to change power options, and the application only allows users with Admin privileges to change the power options for their own account. When a user attempts to change their power options without Admin privileges, they receive an "access denied" error message. Tracing this action led to 5 logged checks out of 1364 total. Two of these logged checks were for \REGISTRY\HKLM\SOFTWARE\Microsoft\Windows \CurrentVersion\Controls Folder\PowerCfg, and we validated that this one ACL was the cause of the least-privilege incompatibility.

From discussing this with internal Microsoft developers we have surmised that power configuration presents difficult policy issues, where certain scenarios call for per-user settings, and yet those settings have machine-wide impact. One user's power option, such as time to turn off hard disks, may interfere with applications running for other logged in users. At the same time, in a shared laptop scenario one could reasonably want the power options to change depending on the person using the machine (e.g., the long-trip user versus the short-trip user). We speculate that it might be possible to better handle this issue by adding a privileged group whose members can change the power configuration. In a shared laptop scenario, one could then add each new laptop user to the group and make a user's configuration dominant when that user is logged in to the console. However, even with this modification, power configuration would still be unavailable to an unprivileged user.

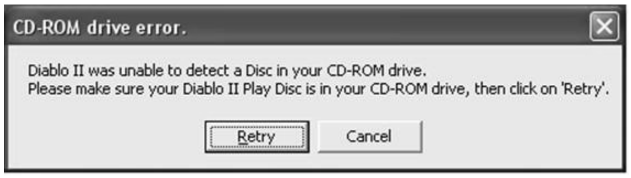
4.2.3 Windows Clock/Calendar

Double clicking the numeric clock on the right-bottom corner of the Windows desktop presents a pictorial clock and a calendar. Users find this a handy tool to use when they want to answer questions like "what is the date of the last Monday of May?" Unfortunately, attempting to launch the clock without Admin privileges leads to an "insufficient privilege to chanhe system time" error message. Even if one wanted to launch this application from a script granting it Admin privileges, this might be difficult because the command line is not readily available. Tracing this action led to 3 logged checks (shown in Table 5) out of 455 total.

We validated that the SystemTime privilege check is the cause of the least-privilege incompatibility. From discussing this case with internal Microsoft developers, we surmised that the original clock was not designed to be used in a read-only manner, but that this privilege check would provide a good place to branch, displaying a read-only UI if the privilege was missing. This least-privilege incompatibility does not appear to fit our model for mitigation by a system administrator.

Security Function	Process	Object Name or Security Action
Access-Check	explorer.exe	\WINDOWS\system32\rundll32.exe
Access-Check	rundll32.exe	\BaseNamedObjects\shell.{A48F1A32-A340-11D1-BC6B-00A0C90312E1}
Adjust-Privilege	rundll32.exe	Enable SystemTime privilege

Table 5. *Log Entries for Windows Clock/Calendar*


CD-ROM drive error.

Diablo II was unable to detect a Disc in your CD-ROM drive.
Please make sure your Diablo II Play Disc is in your CD-ROM drive, then click on 'Retry'.

Figure 3. *Error Message When Starting Diablo II Game without Admin privileges*

4.3 Programmatic Enforcement of Unnecessary Privilege Requirements

The two applications in this section programmatically enforce that the user possess Admin privileges, but they appear to function perfectly well if this check is bypassed. We discuss the reasons for these requirements in more detail in the context of each application.

4.3.1 Diablo II Game

Diablo II is an action game that ships on three CDs: an install disc, a cinematics disc, and a play disc that must be in the drive for the game to work. When a user without Admin privileges attempts to play the game, a misleading error message (Figure 3) pops up claiming the CD drive is empty. Tracing this action generated 3 log entries out of 1573 total checks, 440 of which fail for a user without Admin privileges. The 3 log entries are shown in Table 6.

Because the error message mentions the CD-ROM drive, we hypothesized that the third log entry was responsible for the least-privilege incompatibility. We verified that passing this check alone allows the game to be played without Admin privileges.

This example illustrates how least-privilege incompatibilities can be presented to the user with a misleading error message. We have not received any response from the Diablo II developers, but the misleading error message leads us to believe that the failure mode was not anticipated by the developers. We speculate that this may be a simple programming oversight where the program attempts to acquire certain unnecessary CD-ROM accesses, and that it could easily be fixed. This least-privilege incompatibility also could easily be mitigated by a system administrator using our validation technique.

4.3.2 TurboTax 2003

TurboTax is tax calculation software released by Intuit. Running TurboTax without Admin privileges generates an error message stating that Admin privileges are necessary to use the application. Tracing the application startup generated 11 log entries out of 12503 total security checks. The 11 logged entries break down to one entry for *SID-Compare*, three for *Access-Check* on semaphores, four for *Access-Check* on HKLM registry keys, and three others. Surprisingly, using our validation technique we discovered that just causing the *SID-Compare* call to succeed is sufficient to allow using the application extensively without Admin privileges; we succeeded in running TurboTax, completing a 1040A tax form and printing it to a PDF file. We have not yet received a response from the developers of TurboTax, but we have two different reasons that we believe might have caused the TurboTax developers to insert this check. First, a publicly available transcript of a discussion with an Intuit customer service representative suggests that requiring Admin privileges was a quick fix solution to data privacy concerns [31]. Because Admin privileges convey complete control of the system, leaking information about other users through the application does not represent an increased exposure of private data if the user viewing the information already has Admin privileges. The second reason we considered is that some code path we did not execute generates a failure when the user lacks Admin privileges. On balance, we believe the evidence points to the check for Admin privileges being an explicit decision by the application developers to require Admin privileges.

5 Related Work

A common approach to increasing system security is to sandbox applications or users, so that the scope of individual compromises is decreased. Common sandboxing techniques include virtual machines [15], system call interposition [1, 14, 30], and restricted file systems [12]. Our work differs from this prior art in that we are not inventing a new sandbox or developing a new technology to better implement an existing sandbox. Instead, our tracing technique is designed to help developers and system administrators make use of an existing and well-understood sandbox: the unprivileged user.

Other previous work has investigated technologies for

Security Function	**Process**	**Object Name**
Access-Check	explorer.exe	\Program Files\Diablo II\Diablo II.exe
Access-Check	Game.exe	\REGISTRY\MACHINE\SYSTEM\ControlSet001\Control\MediaProperties\PrivateProperties\Joystick\Winmm
Access-Check	Game.exe	\Device\CdRom0

Table 6. *Log Entries for Diablo II*

building or re-building systems so that they better conform to the principle of least privilege [13, 25, 32]. Provos et al show how separating OpenSSH into privileged and unprivileged parts (privilege separation) would have reduced its vulnerability to several security holes that were later discovered [26]. Brumley and Song describe the Privtrans tool which significantly automates this process using static analysis and annotations on privileged operations [5]. Our technique is complementary to Privtrans, and our tracing technique could potentially be used to automatically produce the annotations required by Privtrans.

A common assumption in much of this earlier work is that some part of the program under investigation (e.g., OpenSSH) legitimately requires the ability to perform a privileged operation. In contrast, our investigation into Windows applications suggests that in many cases, the requirement that the application run in a privileged context is a trivial bug. In other cases, the requirement that the application run in a privileged context reflects a larger design flaw. There was only an argument for the application requiring Admin privileges in two of the eight cases we evaluated, RAS and Power Config.

Some previous research has focused on new models for access control, e.g., Role Based Access Control, Type Enforcement, and Mandatory Access Control [20]. The development of policies for such systems, and in particular the retrofitting of policies to existing applications, has been recognized to require a significant amount of work [28]. A tracing approach likes ours might help in policy development for such systems by providing insight as to why particular applications do or do not violate particular access control policies.

We now consider previous work that has used static analysis [2, 34], a commonly cited alternative to dynamic tracing techniques such as our own. A major strength of static analysis is that it can achieve code-coverage trivially, while dynamic techniques often require sophisticated test-case generation strategies to exercise all code paths, if exercising all code paths is possible at all. However, no one has previously attempted to apply static analysis to this problem, and indeed, we believe static analysis would be difficult or impossible for at least three reasons. First, the underlying property being checked is a function of all ACLs on the system, and this is not a fixed target. It is standard practice to reconfigure ACLs based on the deployment environment [24], and the actual ACLs can depend on runtime state, e.g., virtual directories such as “My Documents” will be mapped to different directories (with different ACLs) depending on the current user. Second, our investigation shows that privilege failures sometimes occur after the flow of control passes through multiple libraries, and static analysis becomes increasingly difficult as the scope of the analysis increases. Lastly, static analysis typically requires source code, and sometimes additional annotations, and so can not be used if portions of the code are only available in binary format. This is a common situation for system administrators, and also for developers due to the common use of third-party components.

The most closely related previous work is the current developer practice of identifying privilege failures by tracing the file system or registry and grepping for ACCESS_DENIED [17]. Our technique goes beyond this by monitoring a complete set of functions within the Windows security subsystem and implementing a more sophisticated noise filtering strategy. Our evaluation in Section 4 justifies the importance of both of these advances for identifying least-privilege incompatibilities. On UNIX systems, system call tracing is sometimes similarly used to debug access failures. Our tracing technique differs from system call tracing in its more sophisticated noise filtering, and the significantly smaller code base that must be correctly understood in order to correctly capture all access failures. System call tracing must monitor all functions that have security implications and are exposed by the OS API, while we only need to monitor five functions in the Windows security subsystem.

A recently proposed alternative approach to eliminating least-privilege incompatibilities is to encourage developers to run without Admin privileges [27]. Initiating this practice helps with code bases being developed from scratch, but is difficult to incorporate into large pre-existing code bases. Even in the development of new software, our tracing technique provides several additional benefits to developers running without Admin privileges: individual test passes can uncover multiple least-privilege incompatibilities; these least-privilege incompatibilities are identified as such (not just as bugs with unknown causes); and additional helpful debugging information is provided, such as complete call stacks (if symbol files are present), object names, and ACLs.

6 Conclusion and Future Directions

Least-privilege incompatibilities cause many Windows users to run with Admin privileges. This significantly increases the vulnerability of Windows systems: any compromise of a user level application becomes a system compromise. To address this problem, we introduce a blackbox tracing technique that identifies the causes for least-privilege incompatibilities. Our technique catches all least-privilege incompatibilties on exercised code paths.

We evaluated our tracing technique using eight least-privilege incompatible applications. These eight applications span a variety of user types, and exhibit a variety of reasons for the underlying least-privilege incompatibilities. Based on these evaluations and subsequent discussions with developers, we conclude that the tracing technique makes fixing or mitigating least-privilege incompatibilities significantly easier.

In the future, our tracing technique would provide even more value if it was integrated with other development technologies, such as the ability to set breakpoints. Also, we believe that our log validation technique highlights the need for a utility to configure security permissions associated with objects other than files and registry keys. In addition to aiding system administrators in mitigating some least-privilege incompatibilities, such a utility would be convenient for developers debugging these incompatibilities. Finally, we speculate that a tracing technique like ours could be fruitfully applied to some least-privilege problems on other operating systems.

Acknowledgements

We thank our colleagues at Microsoft and Microsoft Research for their assistance and insightful comments, and in particular Doug Beck, Bill Bolosky, Brad Daniels, Jon Howell, Jon Pincus, Dan Simon, Helen Wang, Rich Ward, and Alec Wolman.

References

[1] A. Acharya and M. Raje. MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications. USENIX Security 2000.

[2] K. Ashcraft and D. Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. IEEE Security and Privacy 2002.

[3] T. Ball. Abstraction-guided Test Generation: A Case Study, Microsoft Research Technical Report, MSR-TR-2003-86, November 2003.

[4] K. Brown. Keith's Security Hall of Shame. http://www.pluralsight.com/keith/hallofshame/default.htm.

[5] D. Brumley and D. Song. Privtrans: Automatically partitioning Programs for Privilege Separation. USENIX Security 2004.

[6] M. E. Carson. Sendmail without the Superuser. USENIX Security 1993.

[7] CERT. Advisory CA-2004-02 Email-borne Viruses. http://www.cert.org/advisories/CA-2004-02.html.

[8] CERT. AOL Instant Messenger client for Windows contains a buffer overflow while parsing TLV 0x2711 packets. http://www.kb.cert.org/vuls/id/907819.

[9] CERT. Critical Vulnerabilities in Microsoft Windows. http://www.us-cert.gov/cas/techalerts/TA04-212A.html.

[10] CERT. Internet Explorer Update to Disable ADODB.Stream ActiveX Control. http://www.us-cert.gov/cas/techalerts/TA04-184A.html.

[11] H. Chen, D. Wagner, and D. Dean. Setuid demystified. USENIX Security 2002.

[12] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor. SubDomain: Parsimonious Server Security. LISA 2000.

[13] C. Evans. Very secure FTP daemon. http://vsftpd.beasts.org.

[14] T. Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. NDSS 2003.

[15] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. SOSP 2003.

[16] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In Proceedings of the International Symposium on Software Testing and Analysis, pages 53-62. ACM, 1998.

[17] D. GUI. Debugging Permissions Problems. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnaskdr/html/askgui03272001.asp.

[18] M. Howard, J. Pincus, and J. Wing. Measuring Relative Attack Surfaces. Proceedings of Workshop on Advanced Developments in Software and Systems Security, Taipei, December 2003. Also CMU-CS-03-169 Technical Report, August 2003.

[19] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In Proceedings of the International Symposium on Software Testing and Analysis, pages 14-25. ACM, 2000.

[20] P. A. Loscocco and S. D. Smalley. Meeting Critical Security Objectives with Security-Enhanced Linux. In the Proceedings of the 2001 Ottawa Linux Symposium, July 2001.

[21] Microsoft. Certain Programs Do Not Work Correctly If You Log On Using a Limited User Account. http://support.microsoft.com/default.aspx?scid=kb;en-us;307091.

[22] Microsoft. Retrieving the Properties of a Connection (VBScript). http://msdn.microsoft.com/library/default.asp?url=/library/en-us/ics/ics/retrieving_the_ properties_of_a_ connection_vbscript_.asp.

[23] T. Oetiker. MSI Packaging How-to. http://isg.ee.ethz.ch/tools/realmen/det/msi.en.html.

[24] P. Proctor. Hardening Windows NT Against Attack. http://www.secinf.net/windows_security/ Hardening_Windows_NT_Against_Attack.html.

[25] N. Provos. Improving Host Security with System Call Policies. USENIX Security 2003.

[26] N. Provos, M. Friedl, and P. Honeyman. Preventing Privilege Escalation. USENIX Security 2003.

[27] P. Provost. Non-Admin Development in VS.NET 2003. http://www.peterprovost.org/archive/2004/11/01/2040.aspx.

[28] T. Rhodes. FreeBSD Handbook, Chapter 15: Mandatory Access Control, http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/mac.htmlciteseer.nj.nec.com/ganesh03peertopeer.html.

[29] S. Saroiu, S. D. Gribble, and H. M. Levy. Measurement and Analysis of Spyware in a University Environment. NSDI 2004.

[30] M. R. Tal Garfinkel, Ben Pfaff. Ostia: A Delegating Architecture for Secure System Call Interposition. NDSS 2004.

[31] Toups. Administrator Privileges for TurboTax?!?! http://www.dslreports.com/forum/ remark,9732454 mode=flat.

[32] W. Venema. Postfix Overview. http://www.postfix.org/motivation.html.

[33] Y.-M. Wang, R. Roussev, C. Verbowski, A. Johnson, M.-W. Wu, Y. Huang, and S.-Y. Kuo. Gatekeeper: Monitoring Auto-Start Extensibility Points (ASEPs) for Spyware Management. In Proceedings of Usenix LISA, Nov. 2004.

[34] J. Yang, T. Kremenek, Y. Xie, and D. Engler. MECA: an Extensible, Expressive System and Language for Statically Checking Security Properties. ACM CCS 2003.